PYTORCH

Ben

INTRODUCTION

- Pytorch is a deep learning framework that puts Python first.
- Based on Torch but write in Python.
- Tensors and Dynamic neural networks in Python with strong GPU acceleration.

COMPONENTS

torch	a Tensor library like NumPy, with strong GPU support
torch.autograd	a tape-based automatic differentiation library that supports all differentiable Tensor operations in torch
torch.nn	a neural networks library deeply integrated with autograd designed for maximum flexibility
torch.multiprocessing	Python multiprocessing, but with magical memory sharing of torch Tensors across processes. Useful for data loading and Hogwild training.
torch.utils	DataLoader, Trainer and other utility functions for convenience
torch.legacy(.nn/.optim)	legacy code that has been ported over from torch for backward compatibility reasons

FEATURES

- A GPU-ready Tensor library
- Dynamic Neural Networks: Tape based autograd
- Python first
- Fast
- Extensions without pain

GPU ACCELERATION

- Based on NVIDIA CUDA
- Use CUDA semantics then run on GPU
- Support multi-GPU on a single machine.

```
dtype = torch.FloatTensor
# dtype = torch.cuda.FloatTensor # Uncomment this to run on GPU
```

DYNAMIC NEURAL NETWORKS

Most frameworks such as TensorFlow, Theano, Caffe and CNTK have a static view of the world.

- One has to build a neural network, and reuse the same structure again and again.
- Changing the way the network behaves means that one has to start from scratch.

With PyTorch, we use a technique called Reverse-mode auto-differentiation, which allows you to change the way your network behaves arbitrarily with zero lag or overhead.

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```

COMPARISON

- TensorFlow is a safe bet for most projects. Not perfect but has huge community, wide usage.
- I(Fei–Fei Li) think Pytorch is **best** for research. However still new, there can be rough patches.
- Use TensorFlow for one graph over many machines
- Consider Caffe, Caffe2, or TensorFlow for production deployment
- Consider TensorFlow or Caffe2 for mobile

CODE COMPARISON

numpy

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)
w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)
learning_rate = 1e-6
for t in range(500):
  h = x.dot(w1)
  h_relu = np.maximum(h, 0)
  y_pred = h_relu.dot(w2)
  loss = np.square(y_pred - y).sum()
  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h_relu.T.dot(grad_y_pred)
  grad_h_relu = grad_y_pred.dot(w2.T)
  grad_h = grad_h_relu.copy()
  grad_h[h < 0] = 0
  grad_w1 = x.T.dot(grad_h)
  w1 -= learning_rate * grad_w1
  w2 -= learning_rate * grad_w2
```

tensorflow

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = tf.placeholder(tf.float32, shape=(None, D_in))
y = tf.placeholder(tf.float32, shape=(None, D_out))
w1 = tf.Variable(tf.random_normal((D_in, H)))
w2 = tf.Variable(tf.random_normal((H, D_out)))
h = tf.matmul(x, w1)
h_relu = tf.maximum(h, tf.zeros(1))
y_pred = tf.matmul(h_relu, w2)
loss = tf.reduce_sum((y - y_pred) ** 2.0)
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])
learning_rate = 1e-6
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  x_value = np.random.randn(N, D_in)
  y_value = np.random.randn(N, D_out)
  for _ in range(500):
    loss_value, _, _ = sess.run([loss, new_w1, new_w2
                        feed_dict={x: x_value
```

pytorch

```
dtype = torch.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in).type(dtype),
y = Variable(torch.randn(N, D_out).type(dtype),
w1 = Variable(torch.randn(D_in, H).type(dtype),
w2 = Variable(torch.randn(H, D_out).type(dtype)
learning_rate = 1e-6
for t in range(500):
  y_pred = x.mm(w1).clamp(min=0).mm(w2)
  loss = (y_pred - y).pow(2).sum()
  w1.grad.data.zero_()
  w2.grad.data.zero_()
  loss.backward()
  w1.data -= learning_rate * w1.grad.data
  w2.data -= learning_rate * w2.grad.data
```

DEMO

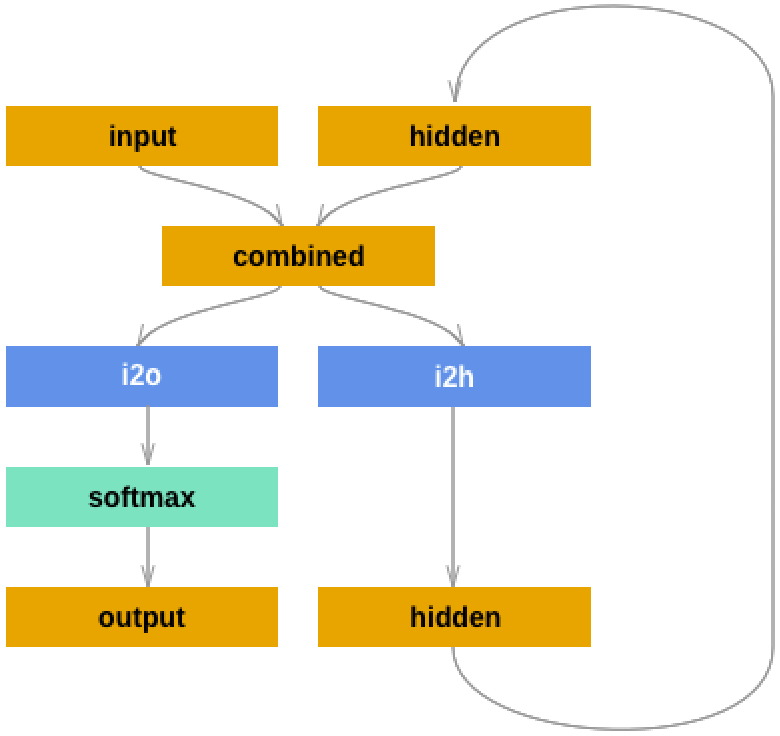
Classifying Names with a Character-Level RNN

- Preparing the Data
- Turning Names into Tensors
- Creating the Network
- Training the Network
- Evaluating the Results

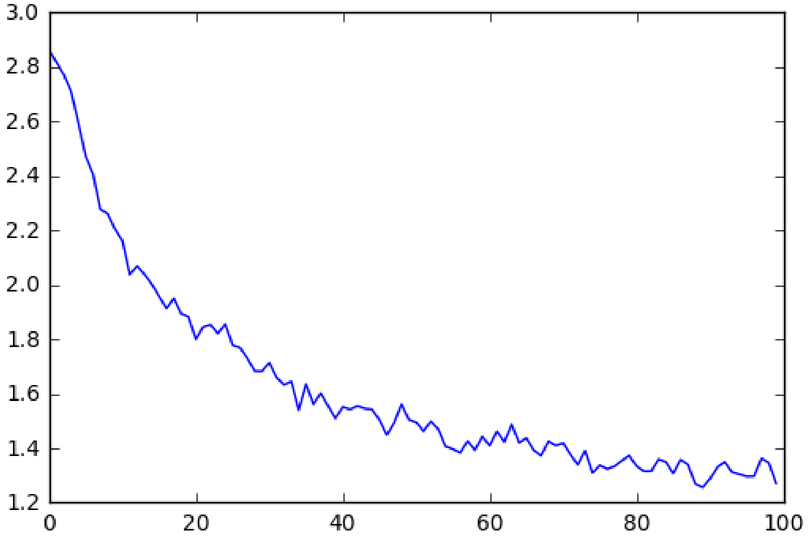
Arabic.txt
Chinese.txt
Czech.txt
Dutch.txt
English.txt
French.txt
German.txt
Greek.txt
Irish.txt
Italian.txt
Japanese.txt
Korean.txt
Polish.txt
Portuguese.txt
Russian.txt
Scottish.txt
Spanish.txt
Vietnamese.txt

```
print(letter_to_tensor('J'))
```

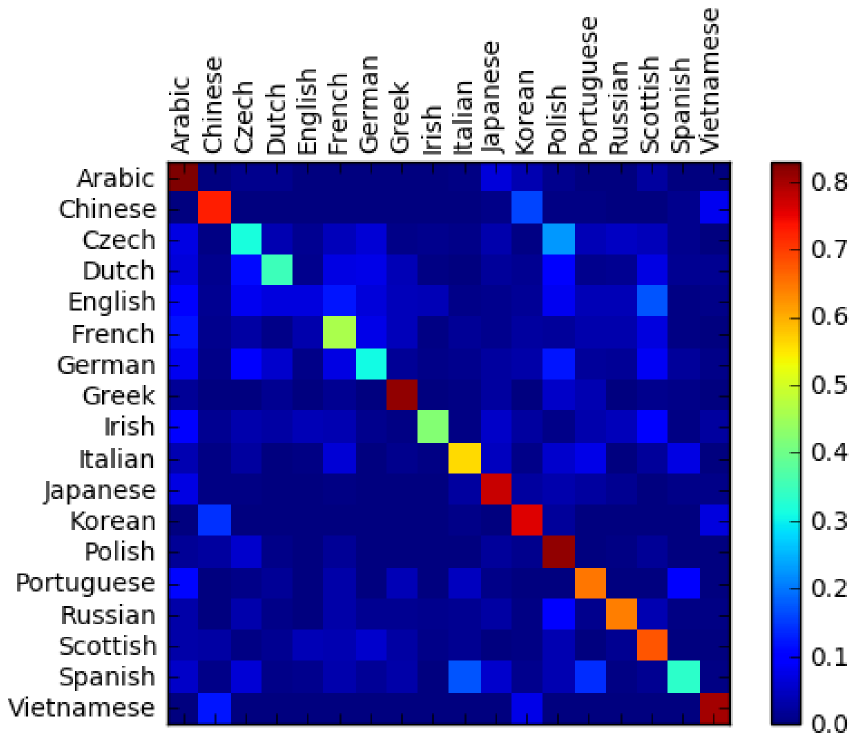
```
Columns 0 to 12
    0     0     0     0     0     0     0     0     0     0     0     0     0

Columns 13 to 25
    0     0     0     0     0     0     0     0     0     0     0     0     0

Columns 26 to 38
    0     0     0     0     0     0     0     0     0     1     0     0     0

Columns 39 to 51
    0     0     0     0     0     0     0     0     0     0     0     0     0

Columns 52 to 56
    0     0     0     0     0
[torch.FloatTensor of size 1x57]
```



RESULT



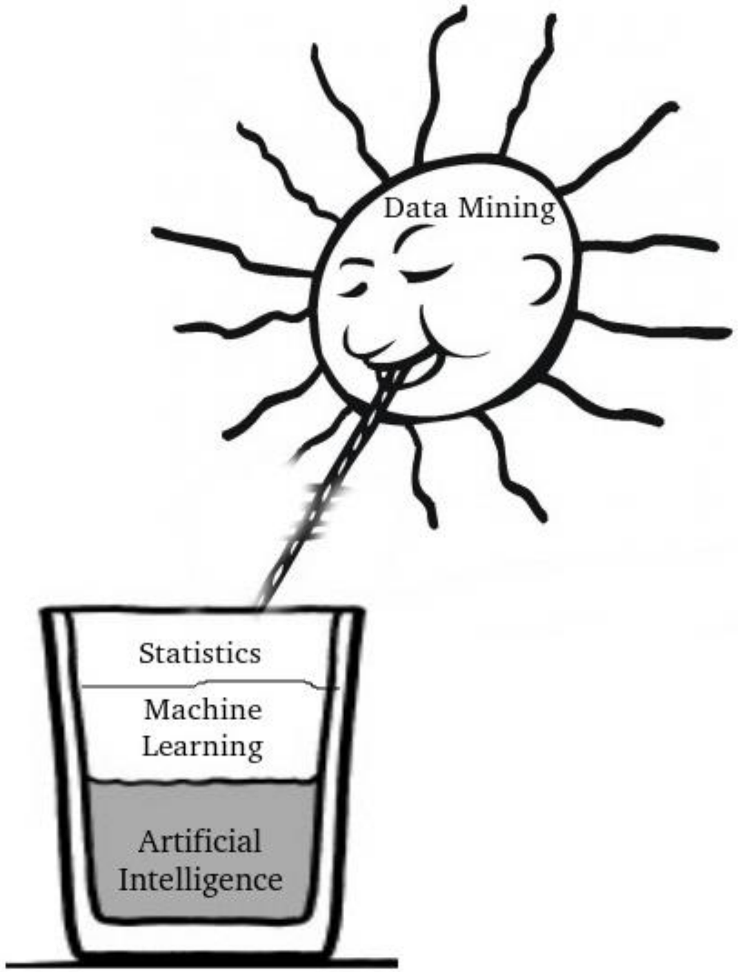

```
> Dovesky
(-0.87) Czech
(-0.88) Russian
(-2.44) Polish

> Jackson
(-0.74) Scottish
(-2.03) English
(-2.21) Polish

> Satoshi
(-0.77) Arabic
(-1.35) Japanese
(-1.81) Polish
```



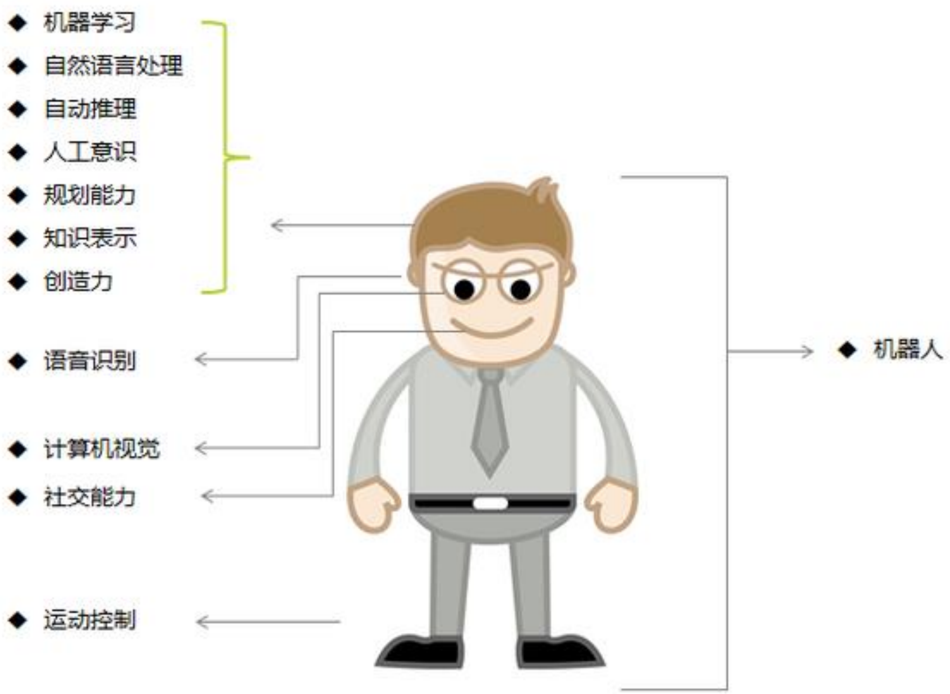
HOW TO STUDY IT

- tutorials
- pytorch/examples
- PyTorch doc
- PyTorch Forums

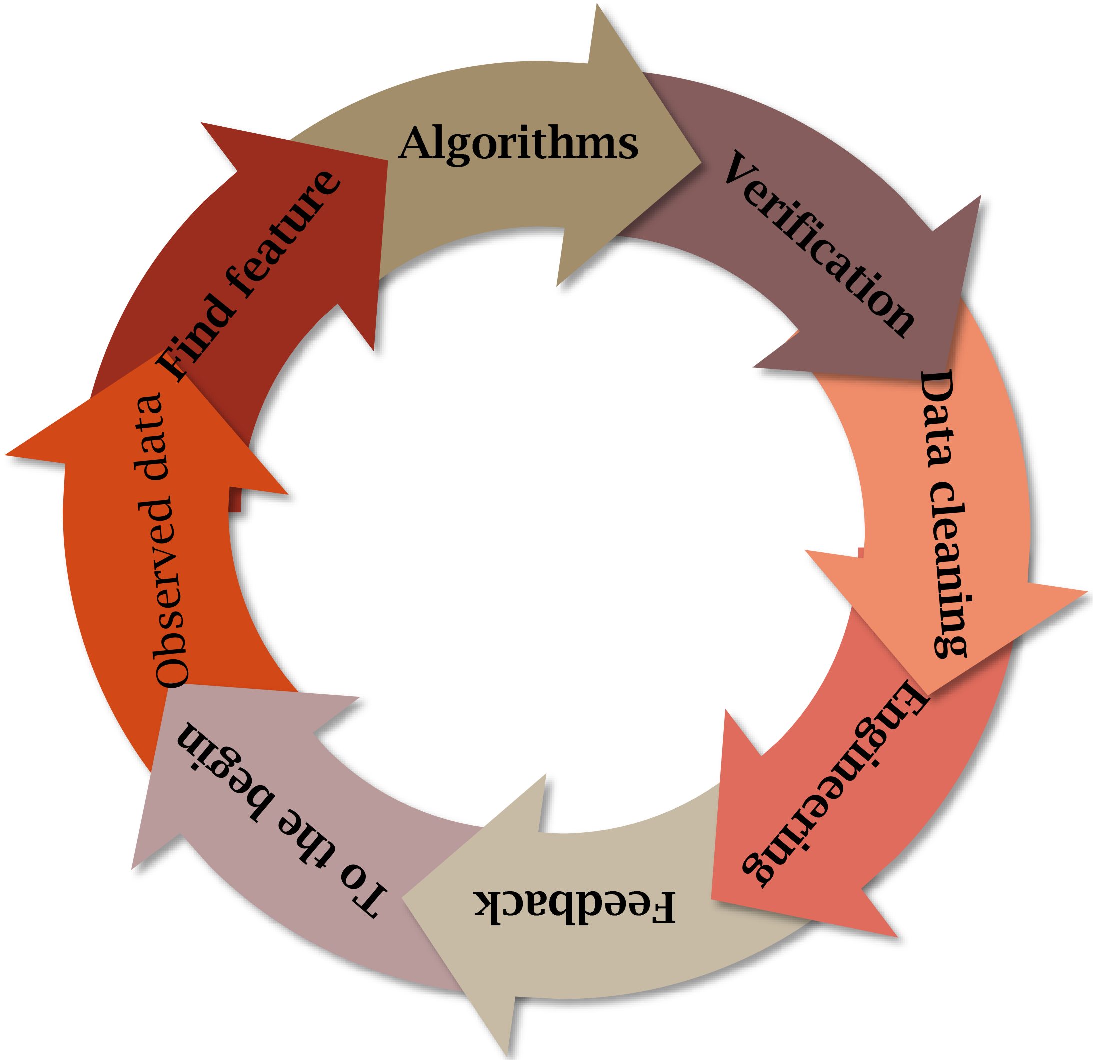
Thanks !


Data Mining
Statistics
Machine Learning
Artificial Intelligence


人工智能
早期的人工智能令人兴奋不已
机器学习
机器学习开始兴起
深度学习
深度学习取得突破
驱动人工智能蓬勃发展
1950's
1960's
1970's
1980's
1990's
2000's
2010's


机器学习
自然语言处理
自动推理
人工意识
规划能力
知识表示
创造力
语音识别
计算机视觉
社交能力
运动控制
机器人

WORKFLOW (APPLICATION)



1. Observed data
2. Find feature
3. Design algorithms
4. Verification
5. Data cleaning
6. Engineering
7. Product feedback
8. To the begin

WORKFLOW (CODING)

A typical training procedure for a neural network is as follows:

- Define the neural network that has some learnable parameters (or weights)
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss (how far is the output from being correct)
- Propagate gradients back into the network's parameters
- Update the weights of the network, typically using a simple update rule:
  weight = weight + learning_rate * gradient